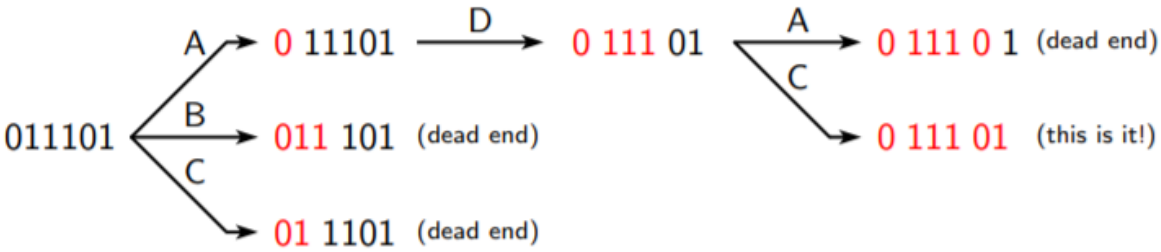
Algoritmos e Estruturas de Dados

Recursively Decoding a Non-instantaneous Binary Code

6 de fevereiro de 2021

Tomé Lopes Carvalho ---------- 97939 → 33.(3)%
Miguel Beirão Branquinho Oliveira Monteiro - 98157 → 33.(3)%
Gonçalo Fernandes Machado ---------- 98359 → 33.(3)%



Introdução

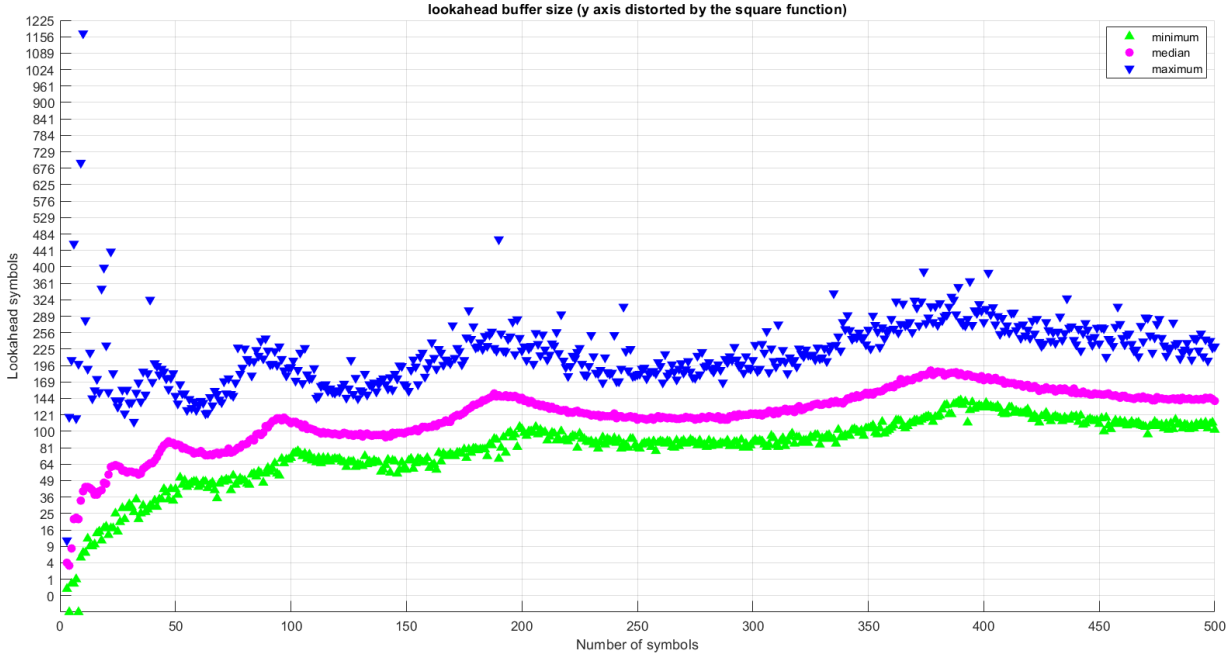
Dado uma mensagem codificada através de um código não ambíguo e não instantâneo e o respetivo dicionário de símbolos - codewords, temos como objetivo descodificar recursivamente a mensagem, explorando todas as interpretações possíveis até atingir um "beco sem saída" (*dead end*) ou chegar à solução. Realizamos esta experiência para n = 3 … 500 (no enunciado pedia apenas até 100, mas achas as amostras demasiado pequenas para análise então realizámos até 500), sendo n o número de símbolos do alfabeto num código, registando a seguinte informação:

- número mínimo, médio, mediana e máximo de chamadas da função recursiva
- número mínimo, médio, mediana e máximo de símbolos descodificados erradamente num caminho

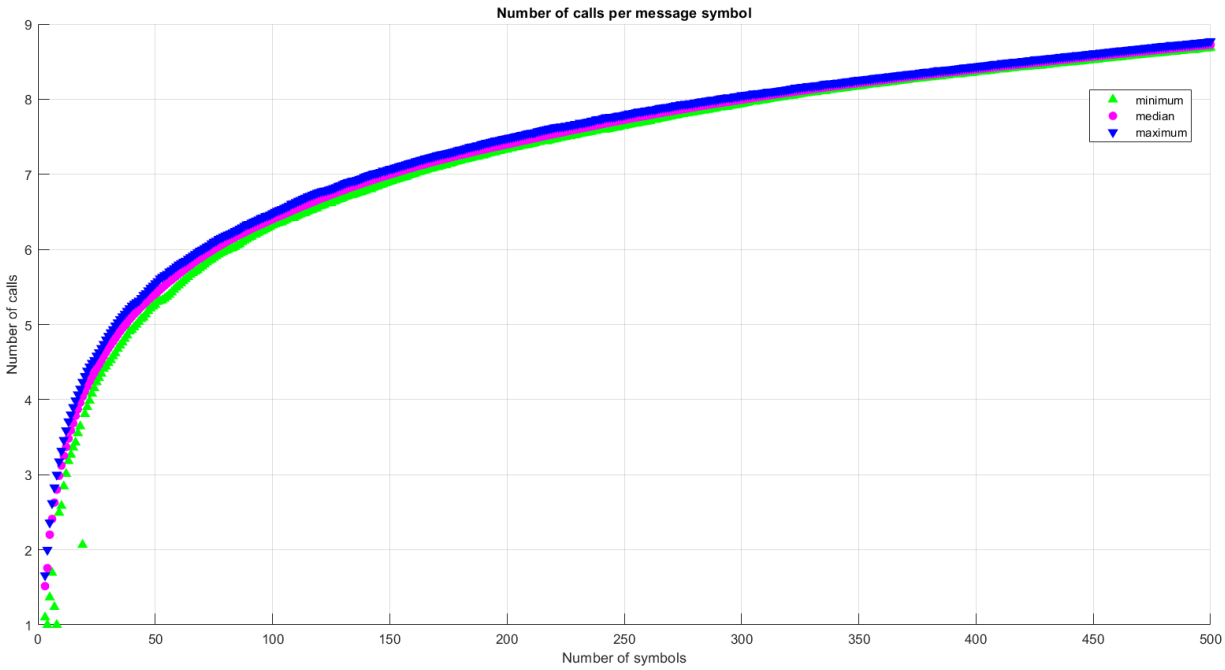
Descrição do Algoritmo desenvolvido - função *recursive_decoder*

1. Incrementar o número de chamadas da função, *_number_of_calls_*;
2. Verificar se estamos no caso base de descodificação correta da mensagem (quando já não há mais bits para descodificar na mensagem). Em caso verdadeiro incrementar *_number_of_solutions_* e fazer *return*;
3. Calcular o número de símbolos descodificados erradamente no caminho através da diferença entre *decoded_idx* e *good_decoded_size*. Caso este valor seja maior que o valor guardado em *_max_extra_symbols_*, atualizá-lo;
4. Para cada símbolo no dicionário:
    4.1. Verificar se pode ser o próximo símbolo da mensagem (p. ex., se a codeword do símbolo for "010", verificar se os próximos bits da mensagem codificada são "010"), utilizando um ciclo *while* que compara bit a bit a codeword com a mensagem codificada
    4.2. Se for possível ser o símbolo, verificar se estamos no caminho certo: *good_decoded_size == decoded_idx* assegura que não cometemos nenhum erro até agora e *i == _original_message_[good_decoded_size]* assegura que é este o símbolo na mensagem original (temos acesso a esta para podermos trabalhar com o *good_decoded_size*);
        4.2.1. Se for o símbolo correto, chamar recursivamente a função incrementando o *encoded_idx* por *j* (comprimento da codeword), *decoded_idx* (número de símbolos descodificados) por 1 e *good_decoded_size* (número de símbolos descodificados corretamente) também por 1
        4.2.2. Se não for o símbolo correto, realizar uma chamada recursiva como a de 4.2.1 mas sem incrementar *good_decoded_size*, uma vez que não o descodificámos corretamente.

Análise dos Resultados



Analisando o gráfico em cima, podemos verificar que existem valores muito elevados de *_max_extra_symbols_*, isto é, o número máximo de símbolos descodificados erradamente até atingir um *dead end*, nos valores iniciais do número de símbolos no dicionário. É possível deduzir que, para um número baixo de símbolos, existem muitas oportunidades para interpretar erradamente os símbolos, causando este elevado número de *_max_extra_symbols_*.
O gráfico é ondulado com picos em valores que aparentam ser 1.5 vezes as potências de 2 (por exemplo, 96, 192 e 384).



Verificamos que é ligeira a diferença entre o máximo, a mediana e o mínimo do número de vezes que a função foi chamada por símbolo, mas à medida que o número de símbolos aumenta, esta diferença diminui. Podemos ainda verificar que os valores evoluem de modo logarítmico.

Função *Recursive_decoder*

```c
static void recursive_decoder(int encoded_idx, int decoded_idx, int good_decoded_size) {
  _number_of_calls_++; // incrementar número de calls da função feitas
  if (_encoded_message_[encoded_idx] == '\0') { // se tivermos descodificado tudo (corretamente, já que não é ambíguo)
    _number_of_solutions_++;
    return;
  }
  // fazer decoded_idx - good_decoded_size, diz-nos o número de símbolos extra que estão mal
  // comparamos a diferença com o max_extra_symbols, se for maior, atualizar max_extra_symbols
  int wrong_symbols = decoded_idx - good_decoded_size;
  if (wrong_symbols > _max_extra_symbols_)
    _max_extra_symbols_ = wrong_symbols;

  for (int i = 0; i < _c_->n_symbols; i++) { // para cada símbolo/codeword
    int possible = 1; // flag que indica se é possível ser este símbolo o próximo
    int j = 0;        // índice do caractere da codeword do símbolo
    // comparar a codeword com a encoded message "bit a bit"
    while (_c_->data[i].codeword[j] != '\0') {               // enquanto não tiver acabado a codeword
      // verificar se o caractere atual coincide com o da encoded message
      if (_c_->data[i].codeword[j] != _encoded_message_[encoded_idx + j]) {
        possible = 0; // se for diferente, é impossível ser este símbolo
        break;
      }
      j++;
    }

    if (possible) { // se puder ser este símbolo
      if (good_decoded_size == decoded_idx && i == _original_message_[good_decoded_size]) { // se for o símbolo correto
(o que está na mensagem original)
        _decoded_message_[decoded_idx] = i; // atualizar a decoded message
        recursive_decoder(encoded_idx + j, decoded_idx + 1, good_decoded_size + 1); // chamada quando estamos no caminho
correto
      }
      else { // se estivermos no caminho errado (sabemos porque fazemos "batota")
        recursive_decoder(encoded_idx + j, decoded_idx + 1, good_decoded_size); // não chamar com good_decoded_size
incrementado
      }
    }
  }
}
```

Matlab Script - Gráfico de Lookahead Symbols

```
clear all; close all; clc;
matriz = zeros(268, 9);
for i = 3 : 270
    file_name = num2str(i, "%.4d");
    matriz(i-2, :) = load(file_name);
end
y = (0:1:36);
hold on;
plot(matriz(:,1), sqrt(matriz(:,6)), 'g^', 'MarkerFaceColor', 'g');
plot(matriz(:,1), sqrt(matriz(:,8)), 'mo', 'MarkerFaceColor', 'm');
plot(matriz(:,1), sqrt(matriz(:,9)), 'bv', 'MarkerFaceColor', 'b');
xlabel("Number of symbols");
ylabel("Lookahead symbols");
title("lookahead buffer size (y axis distorted by the square function)");
yticks(1:36);
yticklabels([0:36].^2);
ylim([0 36]);
grid on;
hold off;
```

Matlab Script - Gráfico de Function Calls per Symbol

```
clear all; close all; clc;
matriz = zeros(268, 9);
for i = 3 : 270
    file_name = num2str(i, "%.4d");
    matriz(i-2,:) = load(file_name);
end
y = (0:1:36);
hold on;
plot(matriz(:,1),matriz(:,2), 'g^', 'MarkerFaceColor', 'g');
plot(matriz(:,1),matriz(:,4), 'mo', 'MarkerFaceColor', 'm');
plot(matriz(:,1),matriz(:,5), 'bv', 'MarkerFaceColor', 'b');
xlabel("Number of symbols");
ylabel("Number of calls");
title("Number of calls per message symbol");
grid on;
hold off;
```